

The EuroDock Simulation Package (ESP) – Technical report

Summary

This report should be considered as an addendum to the report “**Simulation and Visualisation of Docking**” which was written as documentation for the completion of task 5.2 of the EuroDock project. The report is concerned with implementation details of the EuroDock Simulation Package (ESP). Focus is put on three aspects:

- Features added and changes applied to the ESP since the last report.
- Non self-explanatory issues of the ESP. This is stuff that I suspect will be time consuming for third party programmers to extract from the ESP.
- Suggestions to future enhancements/improvements of the ESP.

The report is meant as an internal document to facilitate subsequent maintenance and enhancement of the ESP. As the focus is put on stuff that is incomplete or slightly dubious it may leave the reader questioning the brilliance of the ESP. Needless to say, this is a pitfall that should be avoided.

Additions to the ESP since the subtask report

Since the report for task 5.2 was written, a few changes have been made, mainly concerning the user interface.

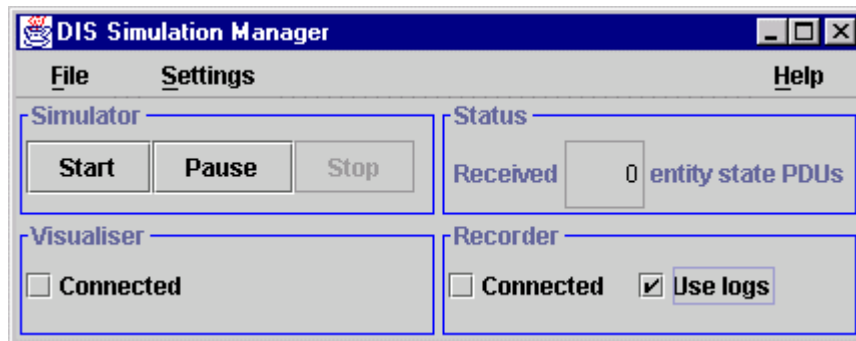


Figure 1: The new user interface.

Recorder interface

The recorder has been completely redefined. Originally the recorder was meant to collect DIS packages and store these in a record-file. Then this file could subsequently be played back at normal, accelerated or reversed speed and the visualiser could be connected as usual but receiving DIS packages from the record file instead of the entities. When the visualizer normally receives DIS packages they are accepted by a Java script (Auv.java) and routed to the VRML scene as events. However, it turned out that visualizing is too slow when events are routed in from a Java script; if events are generated by interpolator nodes in the VRML scene the frame rate could be increased by a factor of 10-20. This can of course not be fixed when concurrent simulation and visualization takes place, but it did lead to a fast and simple design of the recorder where the user can *connect* the recorder the same way the visualiser can be connected. At any time in a simulation the

recorder can be switched on by checking a box on the user interface. When this checkbox later is disabled a VRML file is generated with the startup scene as defined in the Settings menu. Clicking on the surface in the generated VRML file will then start animating all movements of the entities that have taken place during the recording. The recorder uses a skeleton file “Skeleton.wrl” placed in the Java-VRML/Server directory. In the skeleton file the basic structure of the VRML file is stored. Positions and appearance of the entities and environment is set in the Settings menu and the movements are recorded from the DIS packages received during the recording.

The skeleton file is parsed by Server.java using keywords. Any changes in the skeleton file should be made with caution and in awareness that the parser may have to be modified as well.

Log file interface

Another checkbox added to the user interface, controls the use of log files, i.e. the use of files containing pose information for an entity. If the checkbox “Use logs” is checked and a file entityX.tim containing a time vector and corresponding files entityX.pos and entityX.rot containing positions and orientations of entity X can be used to animate this entity. All active entities (as defined in the Settings menu) that have log files present in the Java-VRML directory when a recording is started and “Use logs” is checked will be recorded using these. If a simulation is run when “Use logs” is checked no DIS communication is established with the log file entities. Beware that these entities will not be visualized in a concurrent simulation/visualization. They will only appear in recordings. The log file entities are not DIS entities; they can’t be initialized or communicated with in any way. However, they present an easy way of visualizing entities, which are not yet able to run in the DIS environment.

Compromises and peculiarities

Now the more fishy aspects of the ESP are described. Modularity and consistency are always desirable features but in real life you find you have to make some compromises to make the thing work. In this section the most severe of these are described.

Communication between entities

The simulator as it is built up consists of independent entity models running in different consoles possibly on different computers. Interaction between the entities is therefore a little cumbersome and should be kept to a minimum. In the current simulation scenario there are two communication links between the entities.

- The USBL sensor. The AUV entity has to measure where the DS entity is. For this use a DataQuery PDU is send from the AUV to the DS requiring pose data to be transmitted from the DS to the AUV in certain intervals.
- The GhostDS is an image of where the AUV thinks the DS is. Therefore the AUV controls the pose of the GhostDS by sending out ReferenceData PDUs.

These types of communication suit the DIS concept quite nicely as we have standard PDUs for them and as they are relatively low bandwidth. However, it does bother the modular concept a little as it requires the AUV to know which entity is the DS, and which

is the GhostDS. This is handled by define statements in Pdu.h. Consequently, if you change the order of the entities in the Settings you will have to change these defines too.

End of docking

Entities that closely interact (there is a high bandwidth link between them) does not suit the ESP concept that well. This can occur when two entities physically interact, i.e. a cable connects them, they collide or one of them (say an AUV) docks into another (like a DS). And unfortunately in a EuroDocker simulation such events do take place. When the AUV is inside the docking station it no longer controls its own movements. It is simply dragged around by the DS. These situations could be taken care of by letting the DS send RefData PDUs to the AUV. However, the AUV model would have to be redesigned allowing the dynamics to be bypassed and generating appropriate dead reckoning parameters anyway. Besides, the visualization would still not be very nice as the AUV and DS inevitably would have some relative movement.

Instead the concept of “*gluing*” was introduced. Gluing makes it possible for an entity to signal in the EntityState PDU (using the value 99 in the `SubCategory` field) that it has been glued to the docking station. This is an obvious hack and was never meant as a final solution. Because of that the code only allows entities to attach themselves to the docking entity (entity 1). As gluing is very useful for the visualization I have left it in. It would be a lot better though to define a Data PDU, which an entity can send to signal which entity it glues to (i.e. fixes its position according to) instead of hacking the information through the EntityState. The glue parameter is the only parameter that are “hacked” through PDUs.

Articulated entities

Articulated entities are any movable part of the entity. Here, only the AUV has articulated entities (such as rudders, elevators and cones scaled with the thrust of the propellers). The VRML file that defines the scene ‘Auv.wrl’ –yes the name is very misleading- is meant to be very generic and to constitute a template defining coordinate transforms, environment, entities, light sources and Java script file. The idea was that any inline VRML file could be specified to visualize any of the 5 entities. Ideally then the events controlling the articulated entities should be routed to the inline file and handled there. That would enable programmers to handle articulated entities differently in each entity VRML file (such as Martin.wrl). Unfortunately it was not possible to pass events from Auv.wrl to the entity VRML files. Therefore it was instead decided to make a switch node for each entity with all possible types of articulated entities. The entity model outputs an ID number in the EntityState PDU that uniquely identifies which articulation type to use. This branching on the articulation type also takes place in the Auv.java script. The benefit is that the AUV articulations can be shown for any entity (i.e. all entities can be AUVs). The downside is that when a new articulation type is defined it must be copy-pasted to all entities in the Auv.wrl file. (The file Skeleton.wrl used by the recorder must also be corrected).

Synchronizing time

The time issue has not been completely solved yet. If the Simulator is distributed on different machines the date and time on these machines need to be synchronized. Else, the visualization will not run properly. To alleviate the problem slightly, all entities have been set to start a simulation immediately after receiving a StartResume PDU, and the Server does not filter out packages with “bad” time stamps. Unfortunately this then also means that you can’t send StartResume PDUs with future start dates and you can’t filter out EntityState PDUs in Server.java that have been outdated (have arrived later than a more recent package for the same entity).

Useful utilities

A number of useful utility and test programs are available:

- PduViewer.html: (in the EuroDocker/Simulator/Java-VRML/mil/navy/nps/awt directory). Open a Netscape browser with the correct CLASSPATH variable set (as in StartAll.bat) and the PduViewer allows you to listen on any socket. Using this you can test all communication from the Java user interface, the entities or any other program sending DIS packages that you obtained from the web or coded yourself. The viewer will write out the nature of the PDU and can display all the content of the package as well (indispensable for debugging).
- EspduSender.html: (in the EuroDocker/Simulator/Java-VRML/mil/navy/nps/awt directory). Can send out EntityState PDUs to any socket.
- Gui.mdl (in the EuroDocker/Simulation/Entities directory). A Matlab user interface that can send out SetData, StartResume, StopFreeze (pause), StopFreeze (full stop), and ReferenceData PDUs. The socket number is a parameter to the S-function block. The Gui is very primitive; to use it set exactly one of the PDU flags to one and start and stop the simulation. In the workspace a status message indicating that a PDU was transmitted should appear. If you use this program to transmit to an entity and you use the PduViewer to receive from that entity (set the in port to the server port 36472) then all communication to and from an entity can easily be monitored and tested.

Some comments on notation

The Simulink C-code

Often you encounter a lot of static variables in Simulink C-MEX S-functions. This is in general a bad idea as it can give problems when a model uses multiple instances of the same block. The alternative is to omit static variables and use Simulinks work vectors. This allows Simulink to handle all memory allocation and access and leads to re-entrant code. However, some devious traps are associated with this:

- The number of work vector elements must be specified correctly. If you forget this when you add a variable (and trust me, you will) then Simulink crashes quite unpredictably.
- The variables will have names like iwork[2] which does not reveal much about the function or intended use of the variable. Consequently, the code becomes harder to read and write.
- The syntax for using the work vectors and the input and output vector all differ, which makes mistakes quite likely.

To take care of this problem I've created some accessor macros which standardizes the notation, automatically handles the vector size specifications, allow meaningful naming of variables and further makes reordering, addition and deletion of variables and inputs easy:

```
#define U(element) (*ssGetInputPortRealSignalPtrs(S,0)[element])
#define Y(element) (ssGetOutputPortRealSignal(S,0)[element])
#define I(element) (ssGetIWork(S)[element])
#define R(element) (ssGetRWork(S)[element])
#define P(element) (ssGetPWork(S)[element])
```

Table 1 : Accessor macros.

All input, output and work vectors are then defined using enumerations:

```
enum iworks {
    I_STATE,
    I_STATE_OLD,
    I_STATE_NEXT,
    I_DIMENSION}; /* Always end with I_DIMENSION */
```

Table 2: Defining work vectors.

The variables now have names and are easily accessed for instance by: I(I_STATE). A corresponding input would be accessed by: U(U_STATE). The constant I_DIMENSION at the end of the enumeration is used to specify the length of the vector. So, when the order or number of any input/output or work vector is changed only the enumerations and not the references (or anything else) have to be changed.

Future enhancements

The ESP in its present form is quite generic and can be modified to simulate several scenarios without modifying (recompiling) anything. However, a lot can be improved. The following suggestions would all make the simulator more useful especially for simulating Martin missions such as docking.

Current

Right now current (environment movement) has not been implemented properly. The entities are simply moved graphically. Instead the entities should use the current vector they receive in the SetData PDU and model the influence of the current.

High level commands

The Martin entity currently does not recognize any high level commands. Only RefData PDU for simple movements (like follow a pitch, heading and speed reference or go to a way point) have been implemented. Also HLC commands like Dock-to-entity-X, Abort, Go-to-surface, and Do-a-survey should be implemented.

Pitch regulator

The autopilot on the Martin entity is a simple “home-made” one based on three manually tuned PID controllers. As the dynamics of the vehicle especially in the pitch direction changes with the speed of Martin, this very simple autopilot is not very good in all velocities. A consequence is that docking will fail if Martin is placed with too much

vertical offset to the DS compared to the horizontal distance. A more complex regulator (such as a PID with gain scheduling) should be produced.

USBL model

The USBL model is very primitive; it simply outputs a vector from the AUV to the DS with Gaussian noise added to the three coordinates. Once a real USBL sensor is procured a more accurate model should be made. Additionally, the estimate of the velocity and position of the DS onboard Martin is very simple right now (uses sensor data directly). It should instead be maintained by a Kalman filter or a simple RLS algorithm.

Known bugs

- If the console is chosen from the Settings menu in the user interface, sometimes the user interface graphical appearance can become a little odd. Can be recovered by changing the size slightly (click and drag the lower right corner of the user interface).
- When the Simulink entities are compiled using the RTW some warnings appear due to multiple inclusions of PduCom.c. This is harmless but annoying.
- If you push the start/stop/pause buttons in the user interface too frantically, the state of the user interface may not be the same as the state of the entities (the entities may be running but the user interface indicates that they are not).